APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: SIMULATING A LOGIC DESIGN

APPLICANT: WILLIAM R. WHEELER AND MATTHEW J. ADILETTA

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL 624 273 725 US

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

August 29 2001
Date of Deposit

Samantha Bell
Signature

Samantha Bell
Typed or Printed Name of Person Signing Certificate

SIMULATING A LOGIC DESIGN

TECHNICAL FIELD

This invention relates to simulating a logic design comprised of combinatorial logic and state logic.

BACKGROUND

Logic designs for computer chips typically include combinatorial elements and state elements. Combinatorial elements, such as AND gates and OR gates, combine two or more logic states to produce an output. State elements, such as latches and flip-flops (FFs), hold a logic state for a period of time, usually until receipt of an external clock signal.

Computer languages exist which allow designers to simulate logic designs, including combinatorial and state elements, prior to forming the logic on silicon. Examples of such languages include Verilog and Very High-Level Design Language (VHDL). Using these languages, a designer can write code to simulate a logic design, then execute the code in order to determine if the logic design performs properly.

Standard computer languages may also be used to simulate a logic design.

DESCRIPTION OF THE DRAWINGS

Fig. 1 is a flowchart showing a process for generating simulation code for a logic design.

Fig. 2 is a block diagram of a logic design containing individual state and combinatorial elements.

Fig. 3 is a block diagram of an alternative logic design containing state and combinatorial elements.

Fig. 4 shows a logic cone.

Fig. 5 shows clock domains of a logic design.

Fig. 6 is a block diagram of an alternative logic design containing state and combinatorial elements.

Fig. 7 is a block diagram of a computer system on which the process of Fig. 1 may be performed.

DESCRIPTION

Referring to Fig. 1, a process 10 is shown for simulating a logic design comprised of combinatorial logic and state logic. Process 10 may be implemented using a computer program running on a computer or other type of machine, as described in more detail below.

In operation, process 10 selects (101), in response to a logic designer's input, graphic elements to generate a block

diagram representation of a logic design. The graphics elements are selected from a graphics library and may include combinatorial logic and state logic elements. Graphics elements in the library may be definable or may have predefined functions. For example, the library may contain software objects that perform the function of a flip flop (FF) or a latch. The library may also contain graphics elements that are undefined, i.e., that have no code associated with them.

Each block in the block diagram may represent individual elements or combinations of elements. For example, Fig. 2 shows a graphical representation of a logic design 12 containing combinatorial logic elements 14 to 19 and state logic elements 22 and 24. In logic design 12, each block represents a single combinatorial element (e.g., multiplexers 18 and 19) or state element (e.g., FFs 22 and 24). By contrast, in logic design 26 (Fig. 3), the functionality of several combinatorial elements is contained in a single combinatorial block 28 and the function of several state elements is contained in a single state block 30.

Once the graphical representation of the logic design has been completed (e.g., Figs. 2 and 3), process 10 performs (102) an error check on the design to determine if there are

determines if there are any unterminated or inconsistent connections in the design. If any such problems are detected (103), process 10 issues (104) an error message to the logic designer. The error message may specify the nature of the problem and its location within the logic design. The logic designer is then given the opportunity to correct the problem before process 10 moves forward.

Referring to Fig. 3, process 10 associates (105) computer code that simulates the logic design with each of the graphic elements. For example, process 10 associates computer code with combinatorial logic element 28 to define its function and with state logic element 30 to define its function. The same is true for the logic elements of Fig. 2. The computer code is generated as follows.

Process 10 receives (106) intermediate computer code that is written by the logic designer. The computer code is "intermediate" in the sense that it is an application-specific code, from which simulation code, such as C++ or Verilog, may be generated. In one embodiment, the intermediate computer code includes a netlist that defines logic elements and the interconnections of those elements.

The intermediate code is entered by the designer for each graphic element. For example, the designer may select a graphic element and, using an interactive tool, enter computer code to define the combinatorial (or state) logic represented by that element. The designer may use software objects from the library, such as FFs, latches, AND gates, etc., in the intermediate code. Several objects may be combined within a single graphic element by writing intermediate computer code to effect the combination. For example, an array of FFs may be produced by combining objects from the library via the intermediate code. Graphic element 30 (Fig. 3) contains such an array of FFs.

When designing the logic, the logic designer is constrained by process 10 to represent combinatorial logic and state logic using separate graphic elements. Representing the design using separate state logic and combinatorial logic elements ensures that each separate logic element will achieve its desired state with one iteration, making cycle-based simulation (defined below) possible. Accordingly, process 10 performs (107) an error check to determine if the intermediate code written by the designer includes state logic in combinatorial graphic elements, or vice versa. If a state logic element contains combinatorial

logic or a combinatorial logic element contains state logic, process 10 issues (108) an error message to the designer. The error message may define the problem (i.e., interleaved combinatorial and state logic) and the graphic element that contains the problem. In this embodiment, process 10 requires the designer to correct the problem before process 10 proceeds further. To correct such a problem, the logic designer places the combinatorial logic and the state logic in different graphic elements (i.e., blocks).

Assuming that there are no problems with the design, or that the problems have been corrected, process 10 generates simulation code for the design. In this embodiment, process 10 generates either Verilog computer code or C++ computer code from the intermediate computer code. However, the simulation code is not limited to generating only these two types of simulation code. Any other type of suitable code, an example of which is VHDL, may be generated.

Generally speaking, the designer may select, e.g., via a graphical user interface (GUI) (not shown), which computer code (C++ or Verilog) process 10 will generate. The type of simulation desired may dictate the computer code that process 10 will generate, as described below.

In more detail, two types of logic simulations include cycle-based simulations and event-driven simulations. An event-driven simulation converges on an output of the logic design through multiple cycles. Convergence requires several passes through "logic cones" defined by the computer code.

Referring to Fig. 4, a logic cone 32 is an ordered arrangement in which one logic element 34 passes its output 36 to multiple logic elements 38, which, in turn, pass their outputs 40 to other logic elements 42, and so on. Combining state and combinatorial logic elements within a single logic cone requires multiple passes (iterations) through that cone in order for the state elements to achieve the appropriate states and thus provide the proper output.

The syntax of some simulation languages, such as Verilog, is particularly amenable to event-driven simulations, since they interleave state and combinatorial logic. By contrast, C++ can be used to effect cycle-based simulations. Cycle-based simulations assume that the computer code is ordered correctly, meaning that each logic cone can be traced, with only one iteration, to provide an output. Thus, cycle-based simulations require only a single pass through a logic cone in order to determine its output.

As a result, cycle-based simulators are faster, e.g., an order of magnitude faster, than event-driven simulators (since cycle-based simulators require only one pass, versus multiple passes, through a logic cone). So, for example, on a platform, such as an Intel® Pentium® III microprocessor running at 700 MHz (megahertz), simulating 100 cycles of a complex logic design with an event-driven model might take 10 seconds, whereas performing the same simulation using a cycle-based simulator might take 1 second.

For the foregoing reasons, cycle-based simulations are generally preferred over event-driven simulations. Since separating the combinatorial logic from the state logic facilitates cycle-based simulations, process 10 provides a significant advantage to logic designers.

Referring back to Fig. 1, process 10 decides (109) whether to generate C++ simulation code or Verilog simulation code from the intermediate computer code. This decision (109) is typically made based on an input from the logic designer. If process 10 decides (109) that Verilog code is to be generated, process 10 generates (110) the Verilog code from the intermediate code. The Verilog code may be generated from the intermediate code using a translator program (not shown) and a database (not shown) that

correlates the connections defined by the intermediate code to Verilog code. Parameters or values in the intermediate code are identified and input to the Verilog code.

After process 10 generates (110) the Verilog code, process 10 runs (111) the Verilog code through an event-driven simulator program. The event-driven simulator program runs, and provides inputs (e.g., clock signals), to the Verilog code to generate a simulation of the operation of the logic design. To obtain an output of the logic design using the event-driven simulation, more than one pass through each logic cone defined by the Verilog code may be required.

If process 10 decides (109) to generate C++ code from the intermediate code (based, e.g., on an input from the logic designer), process 10 generates (112a) a topology of each graphic element of the logic design based on the intermediate code. In more detail, process 10 traces the logic gates through the intermediate code for each graphic element in order to determine how the logic gates are connected to one another. Essentially, process 10 obtains a gate topology from the intermediate code.

Process 10 identifies (113) clock domains in the topology. In this context, a clock domain comprises a set of logic elements (gates) that are triggered in response to the

same clock pulse. For example, referring to Fig. 5, logic gates 50 to 60 are all triggered in response to clock pulse 62, which is initially applied to logic gates 50, 51. The intermediate code provided by the designer indicates which clock pulses trigger which logic gates. Accordingly, process 10 traces clock pulses through the logic gates in order to identify the clock domains.

Once the clock domains are identified, process 10 determines (114a) the order in which the logic gates are to be simulated. This is referred to as "code ordering", since the order in which the gates are simulated dictates the order of the resulting C++ code. Process 10 performs code ordering by tracing through each clock domain separately and assigning numerical values to the logic gates.

Referring to Fig. 5, each clock domain 64, 66, 68 can be thought of as a tree having a trunk 70 and branches 72. Process 10 starts at the trunk, in this case logic gate 50, and traverses the tree through to the end of each branch. So, process 10 numbers the trunk (gate 50) "1", then, for branch 74, numbers gate 52 "2", gate 55 "3", gate 56 "4", and so forth. To number another branch, process 10 starts at the trunk and then proceeds in the foregoing manner.

Occasional renumbering may be required, resulting in branches whose gates are not sequentially numbered. This does not present a problem, so long as the assigned number of a child branch is higher than the assigned number of a parent branch. By way of example, assume that there are two starting points (trunks) for clock domain 64. These two starting points are gates 50 and 51. Since both are starting points, they are both assigned number "1". Further assume that branch 72 is first traced starting with gate 51, resulting in gate 51 being assigned a "1", gate 52 being assigned a "2", gate 55 being assigned a "3", and so forth. When branch 72 is retraced starting at gate 50 through path 73, gate 55 is renumbered "4", gate 56 is renumbered "5", and so forth. This may occur as often as necessary in order to ensure that each branch is numbered correctly.

Following the numbering, process 10 examines each clock domain and extracts, from each clock domain, the logic gates numbered "1". These gates are stored in an area of a database. Once this is done, process 10 examines each clock domain and extracts, from each domain, the logic gates numbered "2". These gates are stored in another area of the database. This is repeated then, for each set of logic gates numbered "3", "4", etc., until sets of all numbered logic

gates are stored in different areas of the database. Using this database, process 10 generates simulation code (in this embodiment, C++ code) for the logic gates.

In more detail, for the set of logic gates assigned number "1", process 10 generates C++ code. That is, process 10 defines the connections of the "1" gates, their states, clocks, and other dependencies in C++ code. Following the C++ code for the set of logic gates assigned number "1", process 10 generates C++ code to simulate the set of logic gates assigned number "2". Following the C++ code for the set of logic gates assigned number "2", process 10 generates C++ code to simulate the set of logic gates assigned number "3". This is repeated in sequence until C++ code is generated for all sets of logic gates (e.g., "4", "5", etc.) in the database.

The C++ simulation code may be generated from the intermediate code using a translation program (not shown) and referencing a database (not shown) that correlates the connections and functions specified in the intermediate code to C++ code. Any parameters or values included in the intermediate code are identified and applied to the C++ code.

After process 10 generates the C++ code, process 10 may run (115) the C++ code through a cycle-based simulator

program (this path is indicated by the dotted line in Fig. 1). The cycle-based simulator program provides inputs to, and runs, the C++ code to provide a simulation of the operation of the logic design. To obtain an output of the logic design using the cycle-based simulation, one pass through each logic gate defined by the C++ code is made.

In some instances, a C++ compiler may be unable to compile the C++ code due to its size (i.e., the generated C++ code may be too long for a standard C++ compiler). In these instances, further processing may be required. This processing includes dividing (116) the C++ code into segments based on the numbered logic gates; writing (117) the divided C++ code into separate C++ files and batch files, and compiling the separate C++ files. Thus, in order to use a standard compiler, process 10 compiles C++ code for each set of numbered logic gates. The compiled C++ code may then be run through the cycle-based simulator program separately.

The states of each logic gate are stored in a database as well. Process 10 takes advantage of C++ inheritance capabilities to enable the C++ compiler to handle the large numbers of states that may result from a given logic model. That is, the state of an initial logic gate may be defined as a C++ class. The states of logic gates that depend from the

initial logic gate may refer back to the state of the initial logic gate without actually including data for the state of the initial logic gate. This way, if the state of a subsequent gate depends on the state of a preceding gate, it is possible to obtain the state of the preceding gate without actually adding more data to the database.

Keeping the combinatorial logic and state logic separate according to process 10 makes it possible to identify clock domains and, thus, to perform cycle-based simulations. The advantages of cycle-based simulations are noted above.

Another advantage to keeping combinatorial and state logic separate is that it facilitates manual review of logic designs. Representing the different logic types (e.g., state and combinatorial) in different colors further facilitates the manual review. For example, Fig. 6 shows a logic design 80 that contains both combinatorial logic elements (e.g., 82) and state logic elements (e.g., 84). Simply by looking at logic design 80, it is possible to obtain relevant design information, such as the number of pipeline stages in the design (in this case, there are eight such stages 85 to 92). Other relevant information may also be obtained.

Additionally, by explicitly calling-out state logic elements while in the design environment, it is relatively

easy to develop heuristic tools for providing time/size guidance for a given set of design parameters, such as operating frequency and/or chip area.

Fig. 7 shows a computer 94 for performing simulations using process 10. Computer 94 includes a processor 96, a memory 98, and a storage medium 100 (e.g., a hard disk) (see view 102). Storage medium 100 stores data 104 which defines a logic design, a graphics library 106 for implementing the logic design, intermediate code 108, simulation code 110 that represents the logic design, logic simulator programs 112 (e.g., event-driven and/or cycle-based), and machine-executable instructions 114, which are executed by processor 96 out of memory 98 to perform process 10 on data 104.

Process 10, however, is not limited to use with the hardware and software of Fig. 7; it may find applicability in any computing or processing environment. Process 10 may be implemented in hardware, software, or a combination of the two. Process 10 may be implemented in computer programs executing on programmable computers or other machines that each includes a processor, a storage medium readable by the processor (including volatile and non-volatile memory and/or storage elements), at least one input device, and one or more output devices. Program code may be applied to data entered

using an input device, such as a mouse or a keyboard, to perform process 10 and to generate a simulation.

Each such program may be implemented in a high level procedural or object-oriented programming language to communicate with a computer system. However, the programs can be implemented in assembly or machine language. The language may be a compiled or an interpreted language.

Each computer program may be stored on an article of manufacture, such as a storage medium or device (e.g., CD-ROM, hard disk, or magnetic diskette), that is readable by a general or special purpose programmable machine for configuring and operating the machine when the storage medium or device is read by the machine to perform process 10. Process 10 may also be implemented as a machine-readable storage medium, configured with a computer program, where, upon execution, instructions in the computer program cause the machine to operate in accordance with process 10.

The invention is not limited to the specific embodiments set forth above. For example, process 10 is not limited to simulating only combinatorial and state logic elements. Other logic elements may be simulated. Process 10 is not limited to the computer languages set forth above, e.g., Verilog, C++, and VHDL. It may be implemented using any

appropriate computer language. Process 10 is also not limited to the order set forth in Fig. 1. That is, the blocks of process 10 may be executed in a different order than that shown to produce an acceptable result.

Other embodiments not described herein are also within the scope of the following claims.

What is claimed is: